Cs2100 final cheatsheet

Computer Organisation (National University of Singapore)

CS2100 Computer Organisation

C Programming

-
  ```
  int sumArray(int [], int);
  int sumArray(int arr[], int size);
  ```

-
  ```
  fgets(str, size, stdin) // reads size-1 chars,
      // or until '\n' (then output will have '\n')
  scanf("%s", str);       // reads until white space
  puts(str);  // terminates with newline
  printf("%s\n", str);
  ```

- **Operator precedence**:

Operator	Assoc
`expr++ expr-- () [] . ->`	L to R
`++expr --expr ! ~ (type) * & sizeof`	R to L
`* / %`	L to R
`+ -`	L to R
`<< >>`	L to R
`< <= > >=`	L to R
`== !=`	L to R
`&`	L to R
`∧`	L to R
`|`	L to R
`&&`	L to R
`||`	L to R
`?:`	R to L
`= += -= *- /= %= <<= >>= &= ∧= |=`	R to L
`,`	L to R

- **ASCII values**:

Char	Dec	Hex	Bin
'0'	48	0x30	0b00110000
'A'	65	0x41	0b01000001
'a'	97	0x61	0b01100001

- **Nice numbers**:

$2^{15}-1=$	32 767
$2^{16}-1=$	65 535
$2^{31}-1=$	2 147 483 647
$2^{32}-1=$	4 294 967 295

Number Formats

Integer Formats

- **Sign extension for fixed-point numbers**:
  1's complement: extend sign bit to both left and right
  2's complement: extend sign bit to left and zeroes to right

- **Addition**:
  Perform binary addition. For 1's complement, add the carry-out of MSB to LSB. For 2's complement, ignore carry-out of MSB. If A and B have the same sign but result has opposite sign, overflow occurred. Additionally for 2's complement, if carry-in to MSB $\neq$ carry-out of MSB, overflow has occurred.

IEEE 754 Floating Point Format

	MSB ⟵		⟶ LSB
	Sign	Exponent	Mantissa
Single-precision	**1** bit	**8** bits *(excess-127)*	**23** bits
Double-precision	**1** bit	**11** bits *(excess-1023)*	**52** bits

Instruction Set Architecture

- **Big-endian**: Most sig. byte in lowest address. (MIPS)
  Little-endian: Least sig. byte in lowest address. (x86)

- **Complex Instruction Set Computer (CISC)**
  - Single instruction performs complex operation
  - Smaller program size as memory was premium
  - Complex implementation, no room for hardware optimization

  Reduced Instruction Set Computer (RISC)
  - Keep the instruction set small and simple, makes it easier to build/optimize hardware
  - Burden on software to combine simpler operations to implement high-level language statements

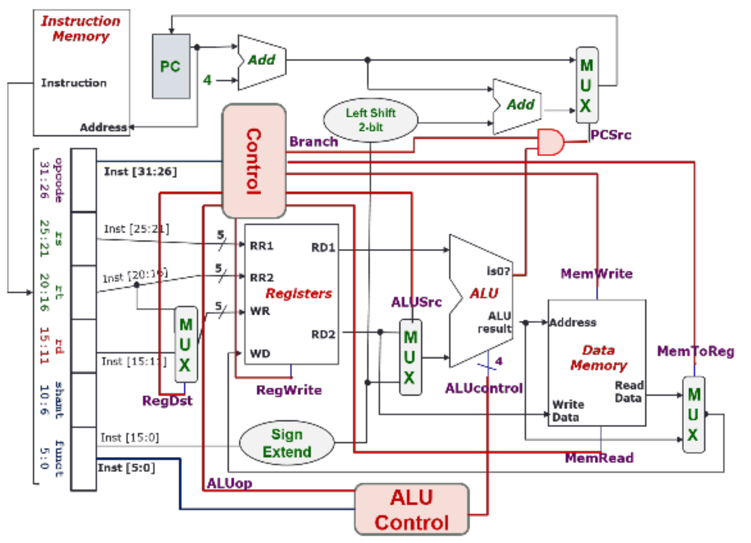
- **Stack architecture**
  - Operands are implicitly on top of the stack.

  Accumulator architecture
  - One operand is implicitly in the accumulator register.

  General-purpose register architecture
  - Only explicit operands.
  - Register-memory architecture (one operand in memory).
  - Register-register (or load-store) architecture.

  Memory-memory architecture
  - All operands in memory. Example: DEC VAX.

The Processor



Datapath & Control

- **Datapath**: Collection of components that process data; performs the arithmetic, logical and memory operations
- **Control**: Tells the datapath, memory and I/O devices what to do according to program instructions

Instruction Execution Cycle

1. Instruction Fetch: Get instruction from memory using address from PC register
2. Instruction Decode: Find out the operation required
3. Operand Fetch: Get operands needed for operation
4. Execute: Perform the required operation
5. Result Write: Store the result of the operation

MIPS combines Decode and Operand Fetch;
MIPS splits Execute into ALU and Memory Access

Clock

- PC is read during the first half of the clock period and it is updated with PC+4 at the

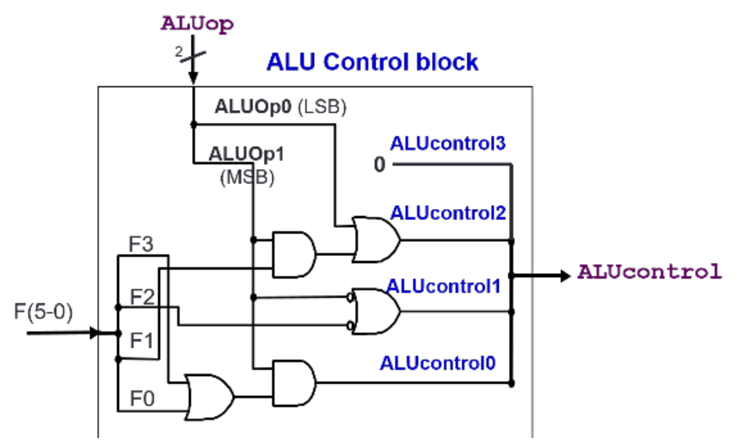
Control Signals

`RegDst`	Decode/Operand Fetch	Select the destination register number
`RegWrite`	Decode/Operand Fetch; RegWrite	Enable writing of register
`ALUSrc`	ALU	Select the 2nd operand for ALU
`ALUControl`	ALU	Select the operation to be performed
`MemRead / MemWrite`	Memory	Enable reading/writing of data memory
`MemToReg`	RegWrite	Select the result to be written back to register file
`PCSrc`	Memory/RegWrite	Select the next PC value

ALUOp Signal (2-bits)

`lw` & `sw`	00
`beq`	01
R-type	10

ALU Control Unit



ALUControl Signal (4-bits, MSB → LSB)

`Ainvert` (1 bit)	Whether to invert 1st operand
`Binvert` (1 bit)	Whether to invert 2nd operand
`Operation` (2 bits)	00 AND | 01 OR | 10 add | 11 slt

Boolean Algebra

Laws & Theorems

Identity	$A+0=0+A=A$	$A\cdot 1=1\cdot A=A$
Inverse/complement	$A+A'=1$	$A\cdot A'=0$
Commutative	$A+B=B+A$	$A\cdot B=B\cdot A$
Associative $A+(B+C)=(A+B)+C$		$A\cdot(B\cdot C)=(A\cdot B)\cdot C$
Distributive $A\cdot(B+C)=(A\cdot B)+(A\cdot C)$		$A+(B\cdot C)=(A+B)\cdot(A+C)$

Idempotency	$X+X=X$	$X\cdot X=X$
One element / Zero element	$X+1=1$	$X\cdot 0=0$
Involution	$(X')'=X$	
Absorption	$X+X\cdot Y=X$	$X\cdot(X+Y)=X$
Absorption (var.)	$X+X'\cdot Y=X+Y$	$X\cdot(X'+Y)=X\cdot Y$
De Morgan's	$(X+Y)'=X'\cdot Y'$	$(X\cdot Y)'=X'+Y'$
Consensus $X\cdot Y+X'\cdot Z+Y\cdot Z=X\cdot Y+X'\cdot Z$		$(X+Y)\cdot(X'+Z)$

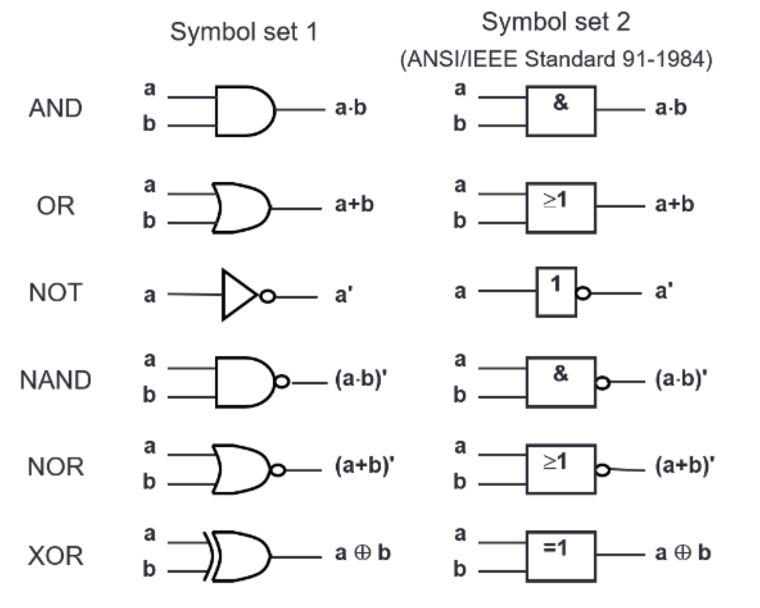
Minterms & Maxterms

- **Minterm**: $\mathrm{m0}=X'\cdot Y'\cdot Z'$
- **Maxterm**: $\mathrm{M0}=X+Y+Z$
- $\mathrm{m0}'=\mathrm{M0}$
- **Sum of minterms**: $\sum \mathrm{m}(0,2,3)=\mathrm{m0}+\mathrm{m2}+\mathrm{m3}$
- **Product of maxterms**: $\prod \mathrm{M}(0,2,3)=\mathrm{M0}\cdot\mathrm{M2}\cdot\mathrm{M3}$
- $\sum \mathrm{m}(1,4,5,6,7)=\prod \mathrm{M}(0,2,3)$

Gray Codes

- Single bit change from one code value to the next
- Standard gray code – formed by reflection

Logic Gates



- **Fan-in**: The number of inputs of a gate.
- **Complete set of logic**: Any set of gates sufficient for building any boolean function.
  e.g. {AND, OR, NOT}
  e.g. {NAND} *(self-sufficient / universal gate)*
  e.g. {NOR} *(self-sufficient / universal gate)*
- **SOP expression** – implement using 2-level AND-OR circuit or 2-level NAND circuit
- **POS expression** – implement using 2-level OR-AND circuit or 2-level NOR circuit
- **Programmable Logic Array (PLA)**: 2-level AND-OR array that can be "burned" to connect

Karnaugh Maps

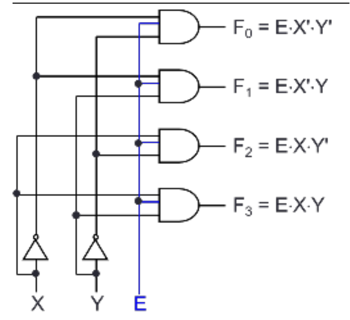
- **Implicant**: Product term with all '1' or 'X', but with at least one '1'
- **Prime implicant**: Implicant which is not a subset of any other implicant
- **Essential prime implicant**: Prime implicant with at least one '1' that is not in any other prime implicant
- **Simplified SOP expression** – group '1's on K-map
- **Simplified POS expression** – find SOP expression using '0's on K-map, then negate resulting expression

Logic Circuits

- **Combinational circuit**: each output depends entirely on present inputs
- **Sequential circuit**: each output depends on both present inputs and state

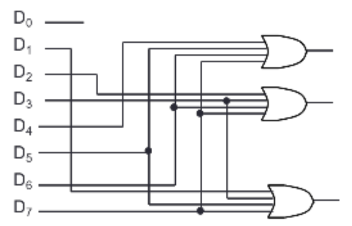
MSI Components

Decoder / Demultiplexer

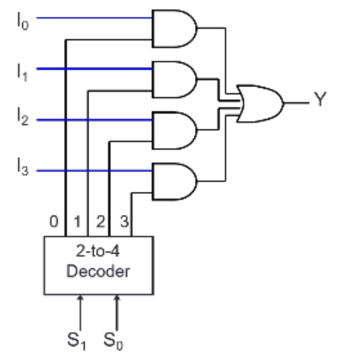


$F_0 = E{\cdot}X'{\cdot}Y'$
$F_1 = E{\cdot}X'{\cdot}Y$
$F_2 = E{\cdot}X{\cdot}Y'$
$F_3 = E{\cdot}X{\cdot}Y$
X
Y
E

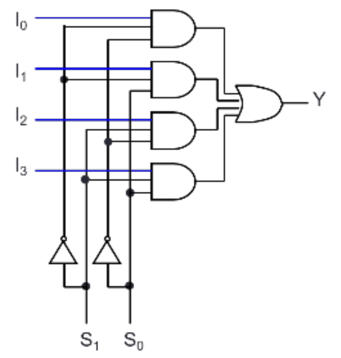
Encoder



Multiplexer 1



Multiplexer 2



- **Decoder** (n-to-m-line decoder): converts binary data from n input lines to $m \le 2^n$ output lines

  Each output line represents a minterm

  Use OR-gates to implement (sum-of-minterms) functions
- **Encoder**: opposite of decoder

  Exactly one input should be '1'
- **Priority encoder**: highest input takes precedence

  All inputs '0' is considered invalid

  Exactly one input should be '1'
- **Demultiplexer**: directs data from input to a selected output line based on n-bit selector

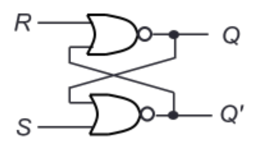
  Demultiplexer $\equiv$ Decoder with enable
- **Multiplexer**: selects one of 2^n inputs to a single output line, using n selection lines

  To implement functions with n variables, pass variables to the n-bit selector and set 2^n inputs to appropriate constants from truth table

  To implement functions with $n+1$ variables, pass first n variables to the n-bit selector and set each input appropriately to '0', '1', Z, or Z' (Z is the last variable)

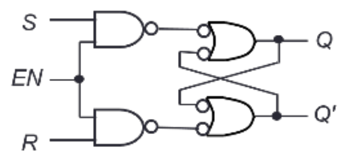
Sequential Logic

- **Synchronous**: outputs change at specific time (with clock)
  Asynchronous: outputs change at any time
- **Multivibrator**: sequential circuits that operate between HIGH and LOW state
  Bistable: 2 stable states (e.g. latch, flip-flop)
  Monostable / one-shot: 1 stable state
  Astable: no stable state (e.g. clock)
- **Memory element**: device that can remember value indefinitely, or change value on command from its inputs
- **Pulse-triggered**: activated by +ve/−ve pulses (e.g. latch)
  Edge-triggered: actv. by rising/falling edge (e.g. flip-flop)
- ***S-R* latch** *("Set-Reset")*:

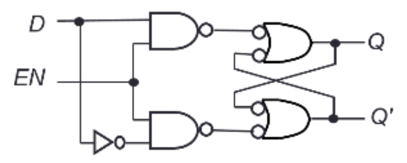


S	R	Q	Q'	
0	0	NC	NC	No change
1	0	1	0	Set
0	1	0	1	Reset
1	1	0	0	Invalid

- **Gated *S-R* latch**: Outputs change only when EN is HIGH

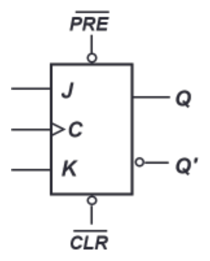


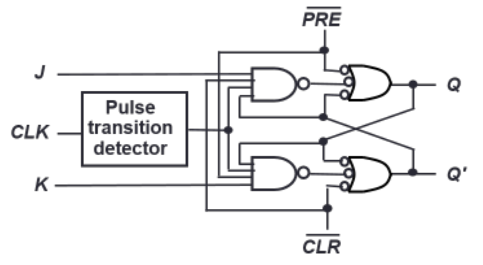
- **Gated *D* latch** *("Data")*: Can build from gated *S-R* latch



No invalid input state

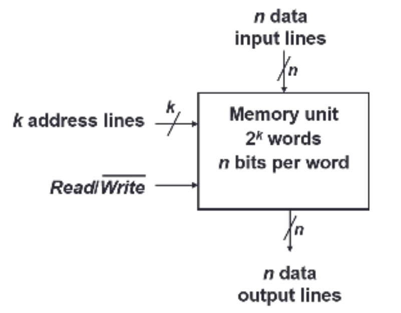
- ***S-R* flip-flop**: Similar to gated *S-R* latch
- ***D* flip-flop**: Similar to gated *D* latch
- ***J-K* flip-flop**: J:*"Set"*, K:*"Reset"*, Toggle if both HIGH
- ***T* flip-flop** *("Toggle")*: *J-K* flip-flop with tied inputs
- **Asynchronous *J-K* flip-flop**: Preset/Clear clock bypass





Memory

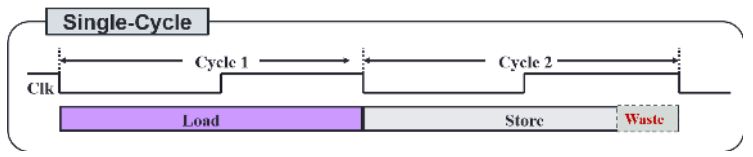
Fast, expensive, volatile $\longleftrightarrow$ *Slow, cheap, non-volatile*
Registers — Main memory — Disk storage — Magnetic tapes

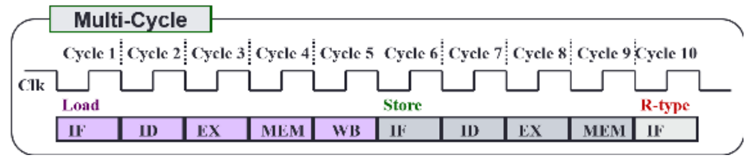


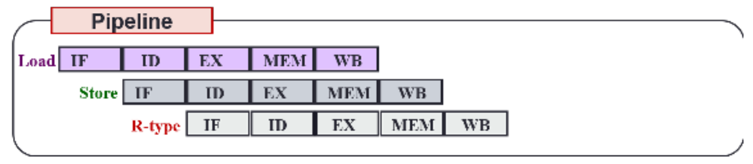
(Optional *memory enable* control signal)

Can be made into arrays using decoder (more words) or parallel (increase word size)

Pipelining







- **Processor Performance**:
  N stages; T_k time required for k^{th} stage; I instructions
  Single-cycle: total time $= I \times \sum_{k=1}^{N} T_k$
  Multi-cycle: total time $= I \times$ (average CPI) $\times \max_{k=1}^{N} T_k$
  Pipeline: total time $= (I+N-1) \times (\max_{k=1}^{N} T_k + \text{(overhead)})$
  (CPI: cost per instruction (number of used stages))
  (overhead: overhead for pipelining, e.g. pipeline register)
- **MIPS Pipeline Stages**:
  IF (Instruction fetch)
  ID (Instruction decode & register file read)
  EX (Execute / address calculation)
  MEM (Memory access)
  WB (Write back)
- **Pipeline registers** between adjacent stages store both data and control signals

Hazards

- **Types of pipeline hazards**:
  Structural: Simultaneous use of hardware resource
  Data: Data dependencies between instructions
  Control: Change in program flow
- **Read-after-write (RAW) dependency** occurs when later instr. reads from register written by an earlier instr.
- **Read-after-write (RAW) data hazard** occurs when later instruction reads from register (strictly) before earlier instruction writes to same register
- **Result forwarding**: happens in pipeline register (in-between stages) to bypass register file; resolves all RAW hazards except `lw`
  - `lw` is resolved via stalling pipeline for one cycle
  - `sw` after `lw` might not need to stall at all
- **Control dependency**: An instruction `j` is control dependent on `i` if `i` controls whether or not `j` executes
- **Reducing stall for branching**:
  Early branch resolution Move branch decision calculation from EX/MEM to ID stage – stall 1 cycle instead of 3 (may cause further stall if reg. is written by previous instruction)
  Branch prediction: Guess the outcome and speculatively execute instructions, if guess wrongly then flush pipeline
  Delayed branch: X instructions following a branch will always be executed regardless of outcome (requires compiler re-ordering of instructions to branch-delay slot(s), or add `nop` instructions)

Caching

Temporal locality: Same item tends to be re-referenced soon
Spatial locality: Nearby items tend to be referenced soon

Different locality for instructions & data

- **Hit rate**: fraction of memory accesses that are in cache
- (avg. access time) =
  (hit rate) $\times$ (hit time) + (1 − (hit rate)) $\times$ (miss penalty)
- **Cache block/line**: smallest unit of transfer between memory and cache
- **Types of misses**:
  Cold/Compulsory: when the block has never been accessed before
  Conflict: same index gets overwritten (direct & set assoc.)
  Capacity: cache cannot contain all blocks (full assoc.)
- **Write policy**:
  Write-through: write data both to cache and main memory
  - using a write buffer to queue memory writes
  Write-back: write data to cache; write to main memory when block is evicted
  - using a "dirty bit" on each cache block
- **Write miss policy**:
  Write allocate: load block to cache, then follow write policy
  Write around: write directly to main memory

Direct Mapped Cache

$\longleftarrow$ Block number $\longrightarrow$		
Tag	Index	Offset

Cache blocks are identified by *Tag*&*Index*, and are stored at location *Index* in the cache

Per-block overhead: Valid flag (1-bit) + *Tag* length
(Initially, all valid flags are unset)

N-Way Set Associative Cache

$\longleftarrow$ Block number $\longrightarrow$		
Tag	Set Index	Offset

A block maps to a unique set of N possible cache locations
- Valid flag + *Tag* overhead for *every* block in set

Fully Associative Cache

$\longleftarrow$ Block number $\longrightarrow$	
Tag	Offset

Block can be placed anywhere, but need to search all blocks

Cache Performance

- **Rule of thumb**: Direct-mapped cache of size N has almost the same miss rate as 2-way set associative cache of size $N/2$
- - Compulsory miss does not depend on size/associativity
  - Conflict miss decreases with increasing associativity
  - Capacity miss does not depend on associativity
  - Capacity miss decreases with increasing size
- **Block replacement policy**
  Least recently used (LRU): the usual policy, hard to track
  First in first out (FIFO)
  Random replacement (RR)
  Least frequently used (LFU)